YTsaurus

Как мы адаптировали динамические таблицы YTsaurus для хранения блобов

Максим Бабенко
Руководитель проекта YTsaurus, Яндекс

13 сентября 2023

Введение

Что такое YTsaurus?

- Основная платформа хранения и обработки данных Яндекса
- Отказоустойчивость
- Масштабируемость
- 12 лет разработки, core-команда из ~50 человек
- Open-source проект! https://github.com/YTsaurus

Типичный крупный кластер YTsaurus в Яндексе

Экзабайты

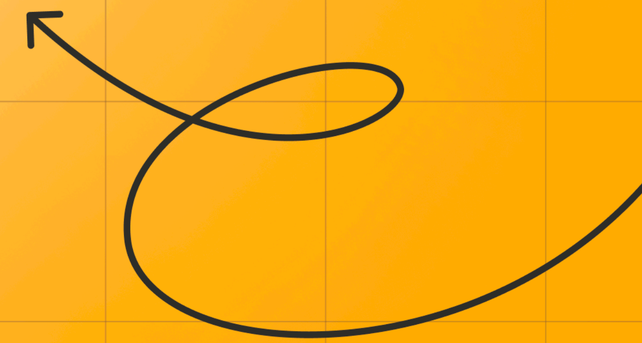
дискового пространства (HDD, NVMe flash)

Десятки тысяч

хостов

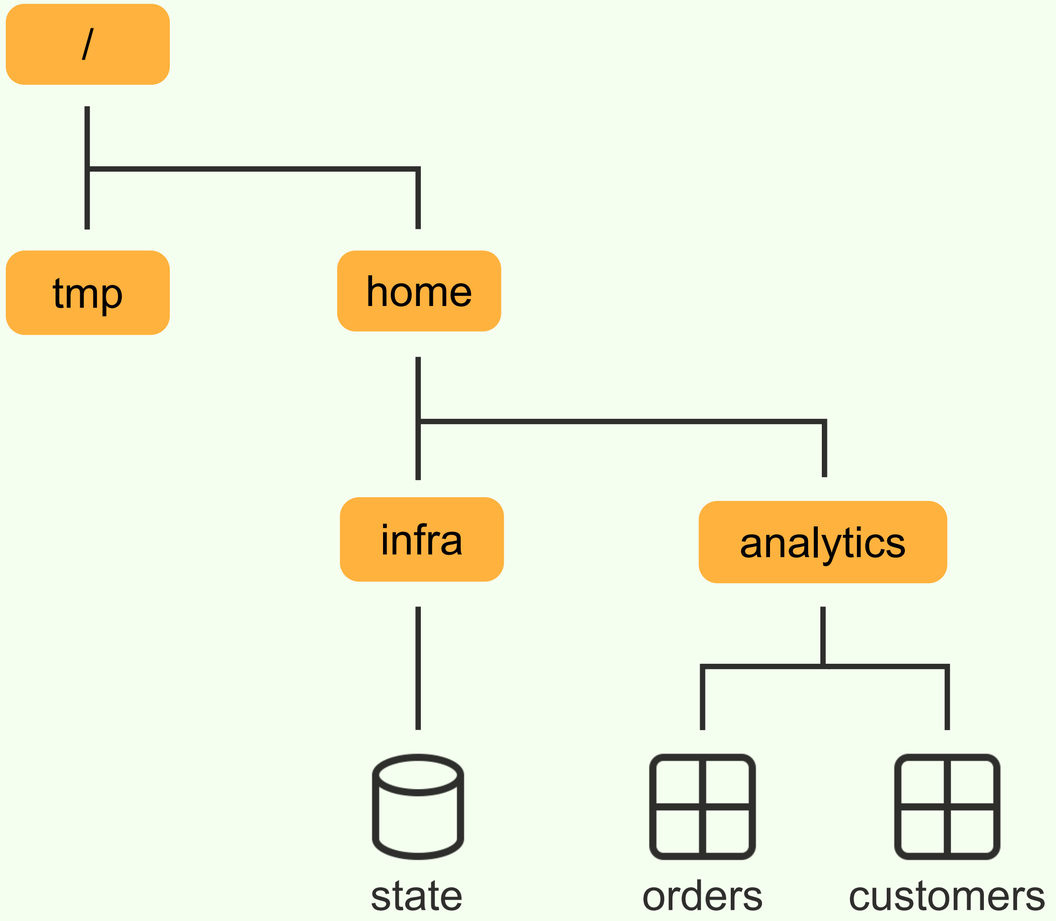
Миллионы

CPU-ядер



Кипарис

- Пространство имен **распределенной файловой системы**
- Древовидная структура
- Внутренние вершины – **каталоги**, листья – **файлы** и **таблицы**



Виды таблиц

Статические и динамические

Виды таблиц

Статические

- Оптимизированы под крупные пачки добавляемых в конец данных
- Эффективный range scan
- Нет точечных обновлений
- Крайне неэффективные точечные чтения

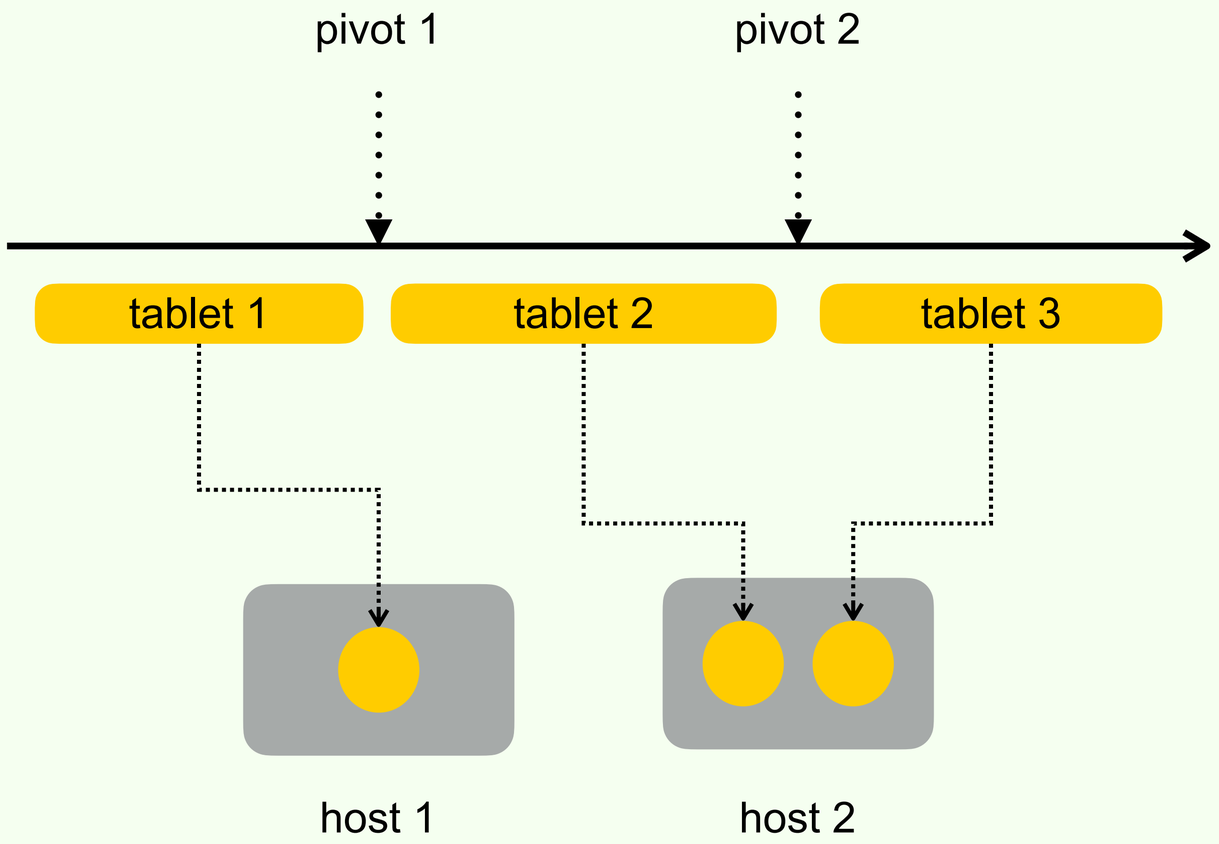
Динамические

- KV-хранилище
- Точечные обновления и чтения
- Range scan (менее эффективный, чем в статических таблицах)
- MVCC
- Распределенные транзакции

Фокус нашего рассказа – на динтаблицах

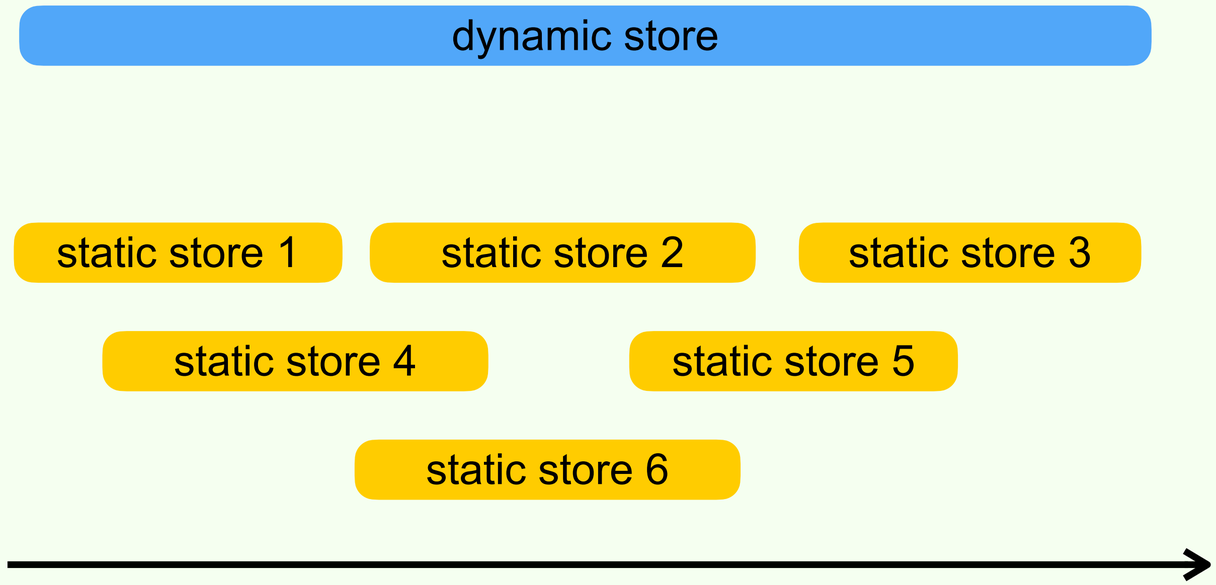
Шардирование динтаблиц

- Пространство ключей разбито на интервалы, задающие шарды (**таблеты**)
- Каждый таблет обслуживается на одном из **хостов** кластера
- **Двухфазный коммит** (2PC) позволяет выполнять распределенные транзакции, затрагивающие несколько таблетов сразу



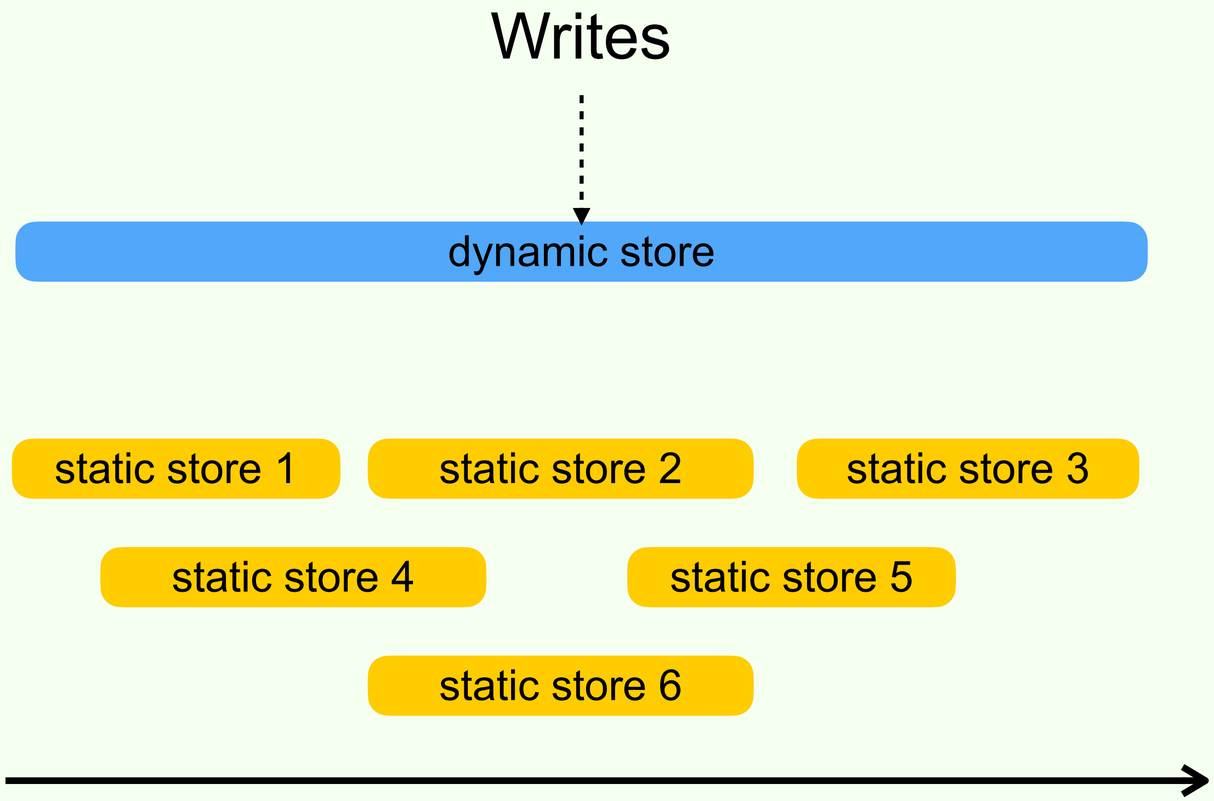
Устройство таблета

- **LSM** (log-structured merge tree)
- Используется в RocksDB, HBase, Cassandra и др.
- Таблет – набор **сторов**
- Сторы: динамические и статические
- **Динамический** стор: мутабельная структура в памяти, принимающая поток изменений
- **Статический** стор**:** иммутабельный блоб на диске, хранящий упорядоченный набор строк



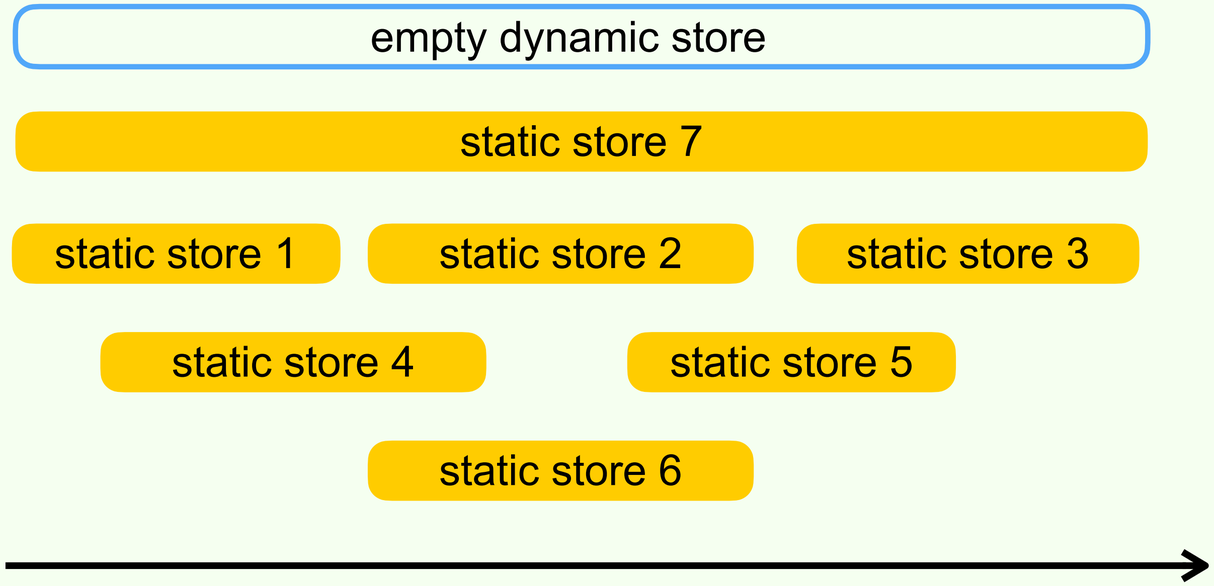
Пайплайн записи

- Входящие данные фиксируются в динамическом сторе



Пайплайн записи

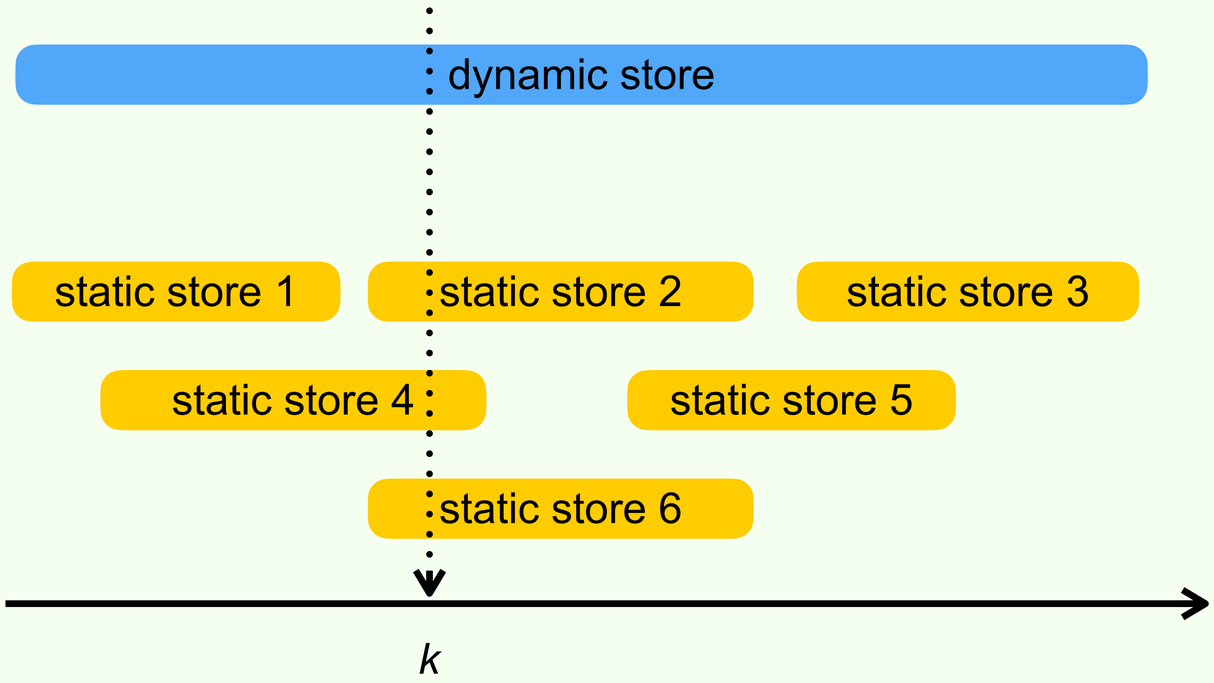
- Входящие данные фиксируются в динамическом сторе
- Динамический стор периодически **сбрасывается** на диск и превращается в статический
- Только **линейная последовательная крупногранулярная** запись на диск (в отличие от B-деревьев!)



Пайплайн чтения

- При чтении таблета по ключу *k* необходимо отобрать все сторы, которые потенциально могут содержать *k*
- Нужно выполнить точечное чтение в них (можно параллельно)
- Нужно выбрать **самую свежую** версию данных

Версии данных: монотонные таймстемпы, hybrid logical clock



Компатификация

Амплификация при чтении

- Сторов в таблете может быть много
- Чтение объема *X* из таблицы приводит к чтению объема *RX* с диска
- *R* – **коэффициент амплификаций** при чтении
- Часть сторов можно пропустить, глядя на их **граничные ключи**
- Часть сторов может пропустить, обратившись к B**loom-фильтру** (закешированному в памяти)
- Амплификация пропорционально **толщине** накрытия пространства ключей сторами
- Значения *A*~5 вполне типичны

Компактификация

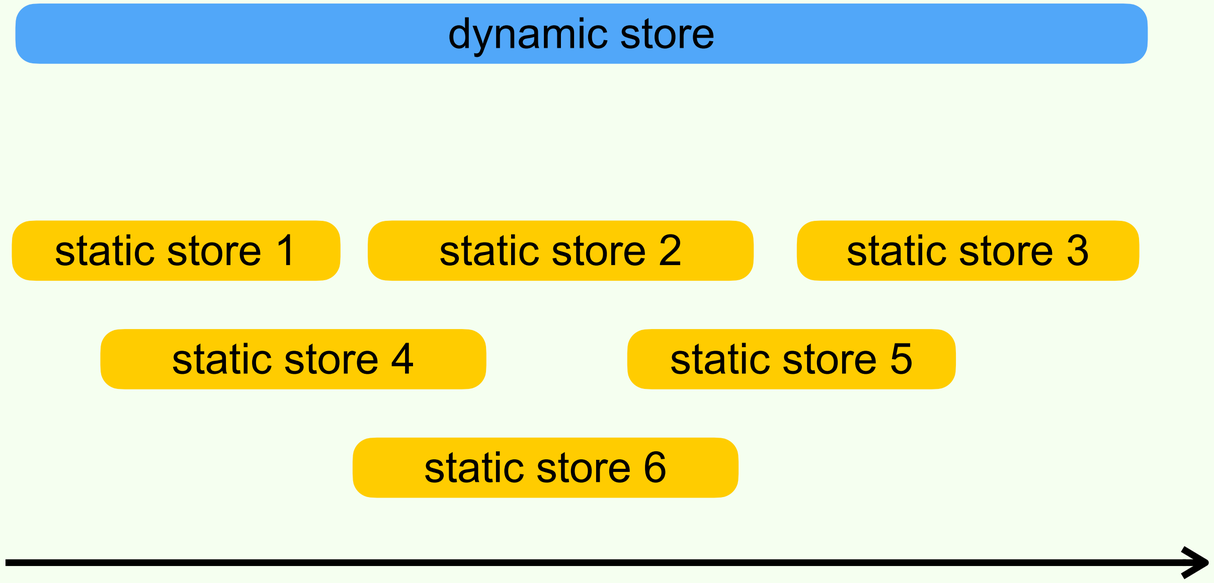
- **Компактификация**: периодически набор выбираем подмножество сторов, сливаем их, режем на части и заменяем на результат слияния
- Компактификация ограничивает **толщину накрытия**
- Выбор сторов для компактификации: сложный **эвристический** алгоритм

Амплификация при записи

- С учетом будущих компактификаций запись объема *X* в таблицу приводит к записи объема *WX* на диск
- *W* – **коэффициент амплификаций** при записи
- Помимо компактификации, нужно также учесть **репликацию** и **write-ahead logging**
- Итоговый полный *W* легко может быть >10

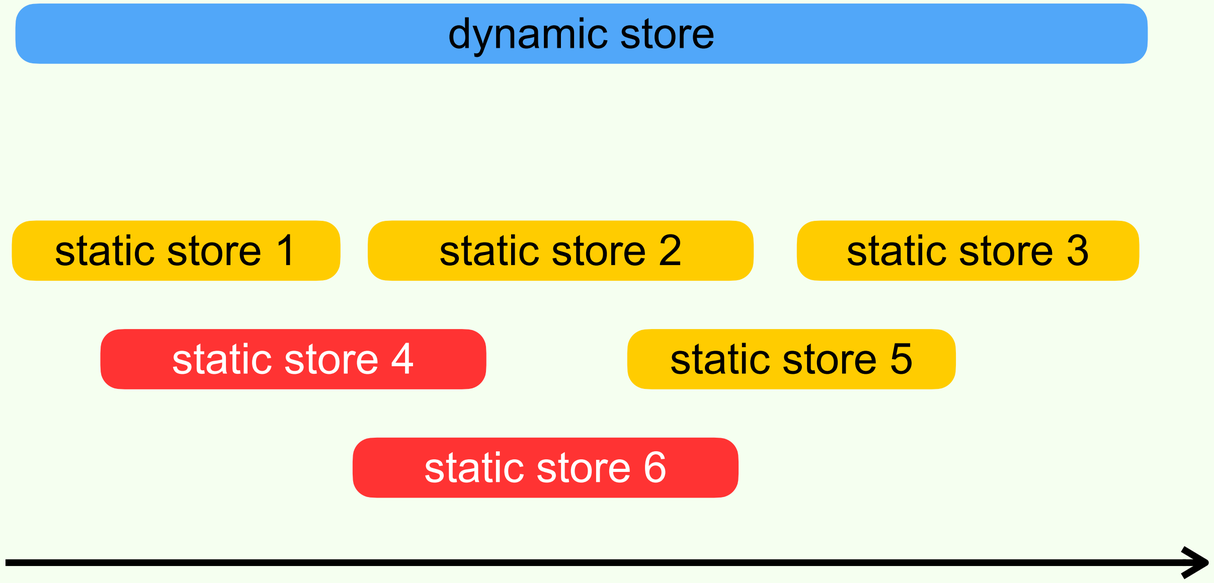
Пример работы компактификации

- В данном случае толщина накрытия пространства ключей статическими сторами **равна 3**



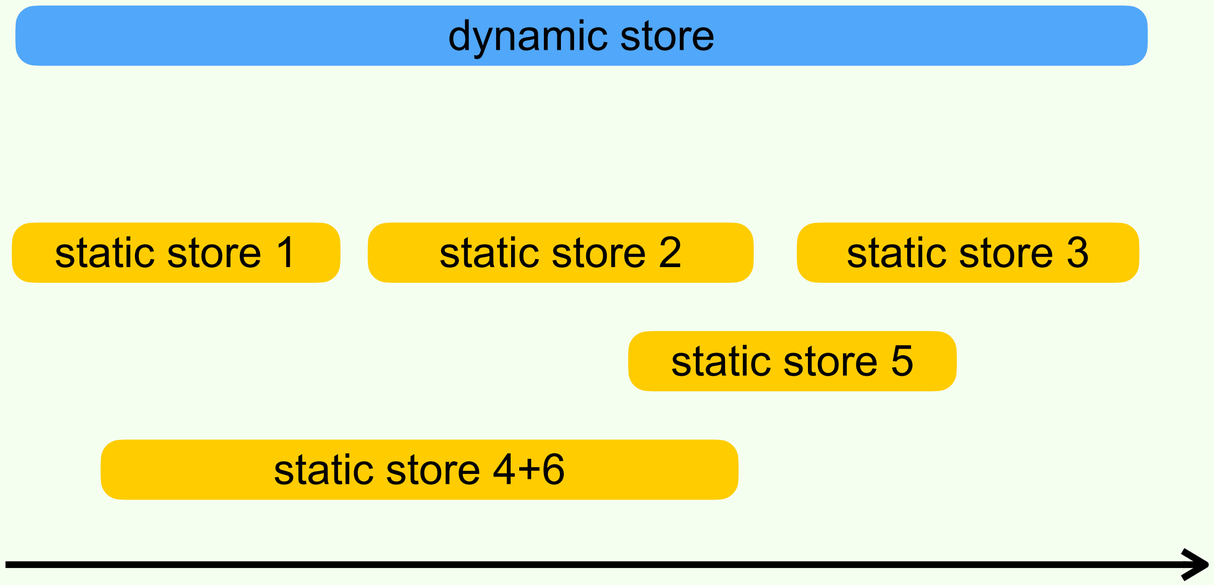
Пример работы компактификации

- В данном случае толщина накрытия пространства ключей статическими сторами **равна 3**
- Возьмем **сторы 4 и 6** и скомпактифицируем их



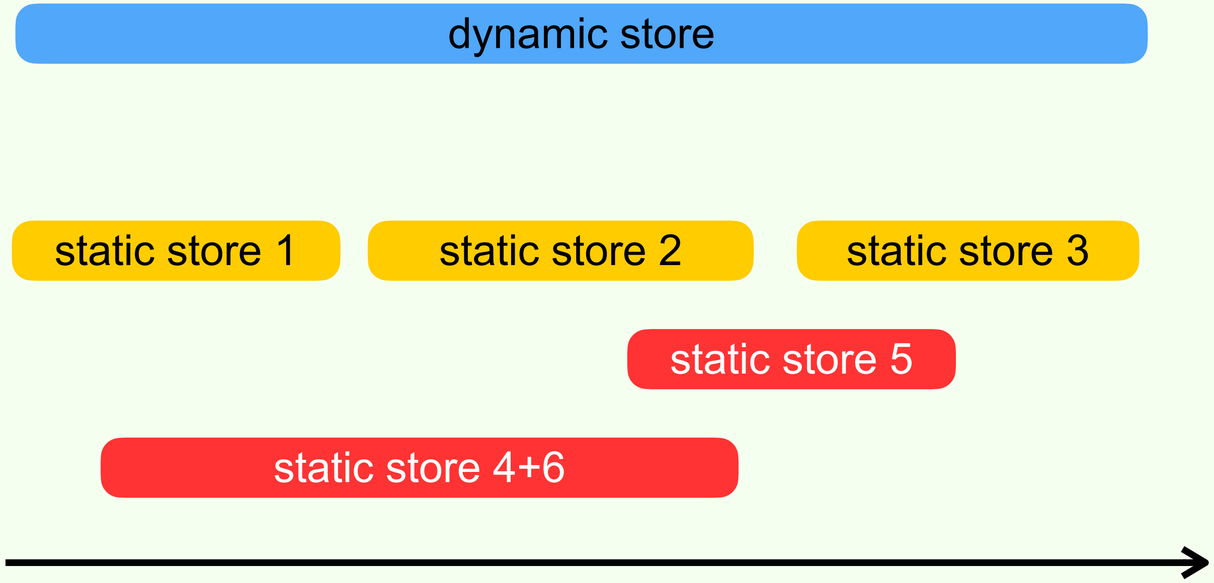
Пример работы компактификации

- В данном случае толщина накрытия пространства ключей статическими сторами **равна 3**
- Возьмем **сторы 4 и 6** и скомпактифицируем их
- Стало ли лучше? Не особенно, толщина накрытия **все еще 3**



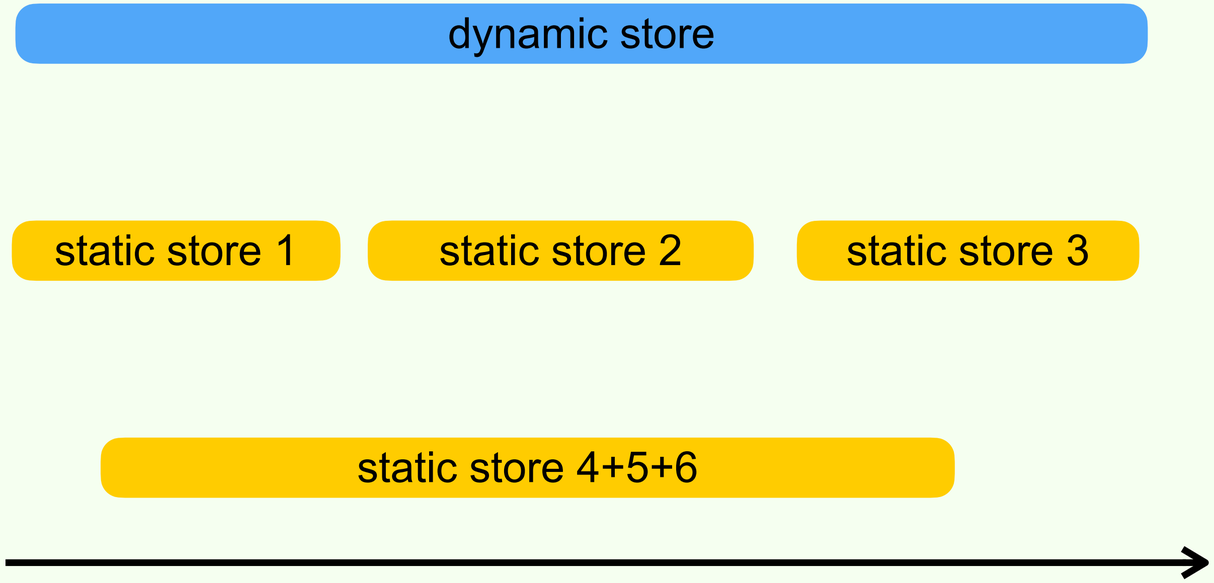
Пример работы компактификации

- В данном случае толщина накрытия пространства ключей статическими сторами **равна 3**
- Возьмем **сторы 4 и 6** и скомпактифицируем их
- Стало ли лучше? Не особенно, толщина накрытия **все еще 3**
- Добавим в компактификацию еще **стор 5**



Пример работы компактификации

- В данном случае толщина накрытия пространства ключей статическими сторами **равна 3**
- Возьмем **сторы 4 и 6** и скомпактифицируем их
- Стало ли лучше? Не особенно, толщина накрытия **все еще 3**
- Добавим в компактификацию еще **стор 5**
- Толщина **упала до 2**!



Выбор сторов для компактификации: сложный **эвристический** алгоритм

Ханки

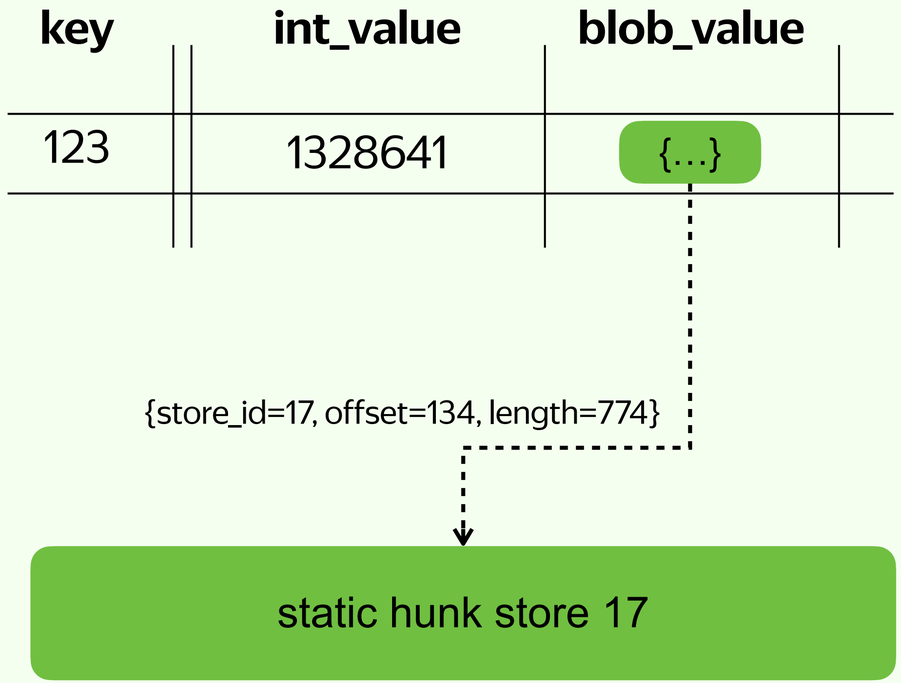
Боремся с амплификацией записи

Проблема амплификации записи для блобов

- Пусть в таблице одна из колонок хранит **крупные блобы** (string, YSON/JSON)
- Компактификация будет **постоянно переписывать** эти блобы, копируя из между сторами на диске
- Можно ли сделать **лучше**?

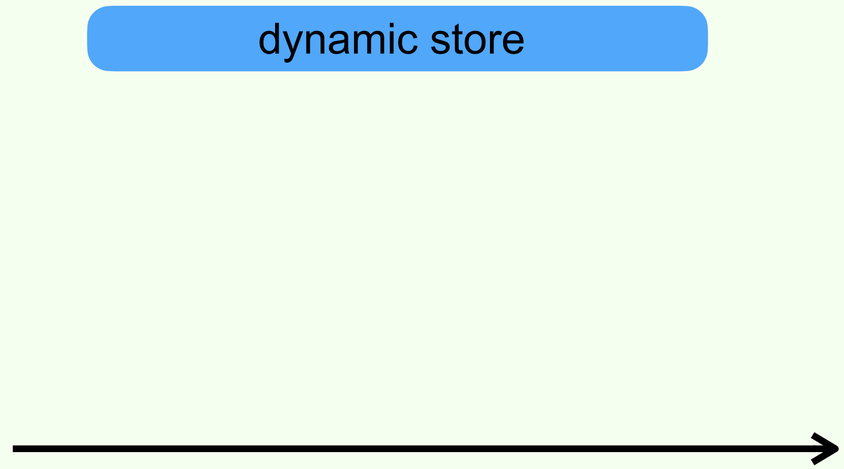
Ханки (hunks)

- **Ханк** – это блоб в ячейке таблицы
- Можно сложить ханки табличного ханки из стора в отдельный **ханковый** стор
- В табличном сторе остаются короткие **ссылки** на ханки (~10 байт, id ханкового стора, смещение в нем, длина)
- Не обязательно заменять ссылками все ханки, можно лишь **достаточно большие**



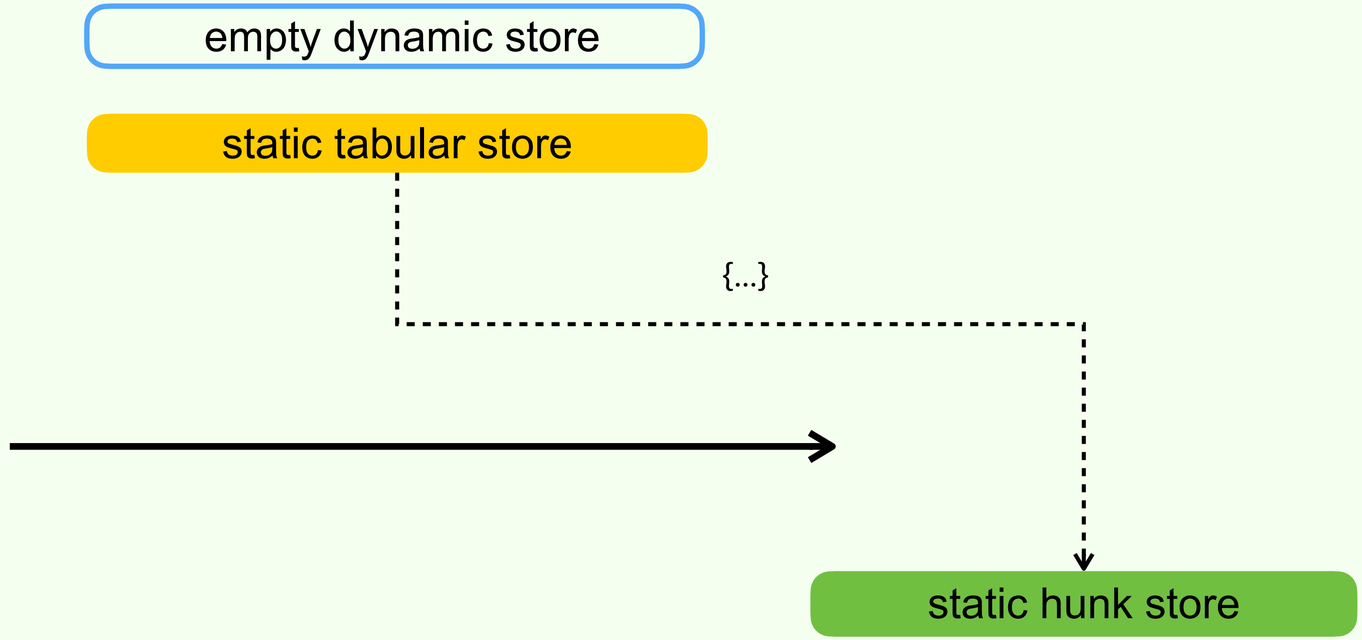
Пайплайн записи при наличии ханков

- При сбросе динамического стора на диск получается два стора



Пайплайн записи при наличии ханков

- При сбросе динамического стора на диск получается два стора: **табличный** (*метаданные*) и **ханковый** (*данные*)
- При компактификации мы заменяем лишь табличные сторы, переиспользуя при этом ханковые
- Амплификация при записи **падает**: компактификация переписывает лишь короткие ссылки на ханки

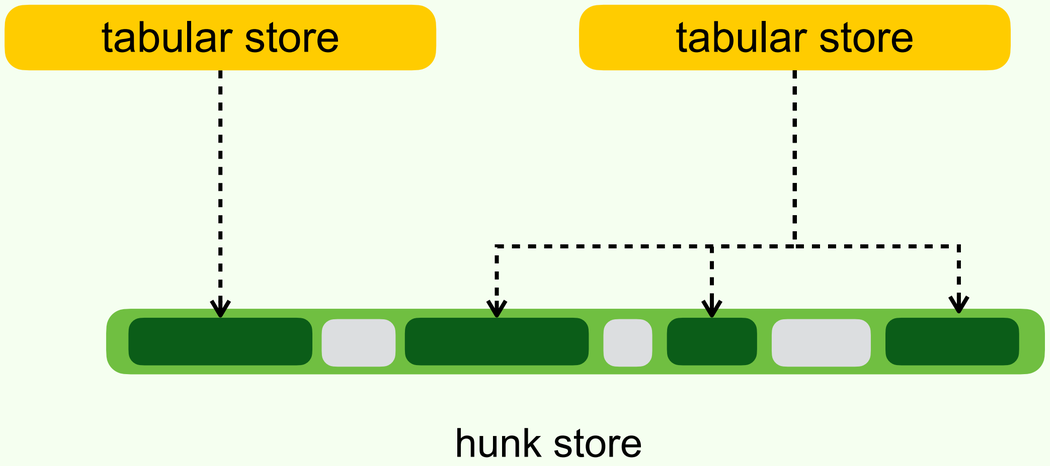


Еще одна амплификация

- **Амплификация по объему** *S*: насколько больше данные занимают на диске по сравнению с их *истинным размером*
- Может ли быть *S* > 1? Да, LSM хранит **множество версий** одной и той же ячейки
- Компактификация может удалять **старые версии** (в зависимости от настроек)

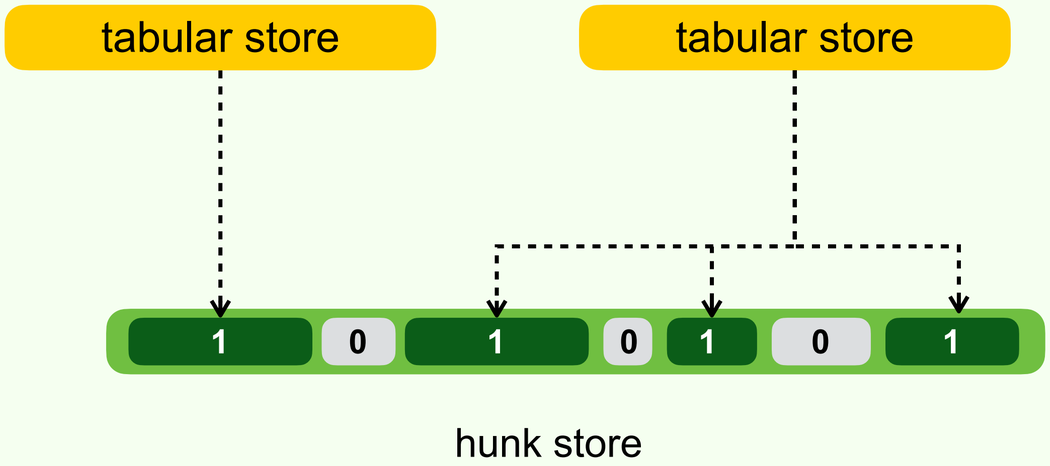
Ханки и мусор

- Изложенная схема никогда не удаляет ханковые сторы, даже если данные в них никому **не нужны**
- В ханковых сторах копится **мусор**
- *S* растет **неограничено**
- Как **собрать мусор?**



Первые мысли

- Давайте помнить для каждого ханка количество **ссылок** на него
- Когда ссылка появляется – **увеличиваем** счетчик, когда пропадает – **уменьшаем**
- Если счетчик **нулевой,** то ханк мусорный, **выбросим** его



Почему это не лучшая идея

- Счетчики нужно где-то помнить, их много, нужна **параллельная структура данных** (в памяти и на диске)
- Как удалить ханк из **середины** ханкового стора?
- Ханковые сторы **иммутабельны**, вырезать середину нельзя
- Даже если бы было можно, эта операция **инвалидирует ссылки** на ханки (меняются смещения)
- Можно сделать **новый ханковый стор**, но это опять **инвалидирует ссылки** (меняется id стора)
- Делать **новый ханковый** стор всякий раз, когда появился один мусорный ханк -- **бесконечно эффективно**

Собираем мусор

Батчинг удаления

В ханковом сторе **много** мусорных ханков, то сделать новый ханковый стор без них уже не выглядит безумием

Остаются проблемы **обнаружения мусора** и **инвалидации ссылок** на ханки

Как опознать мусор?

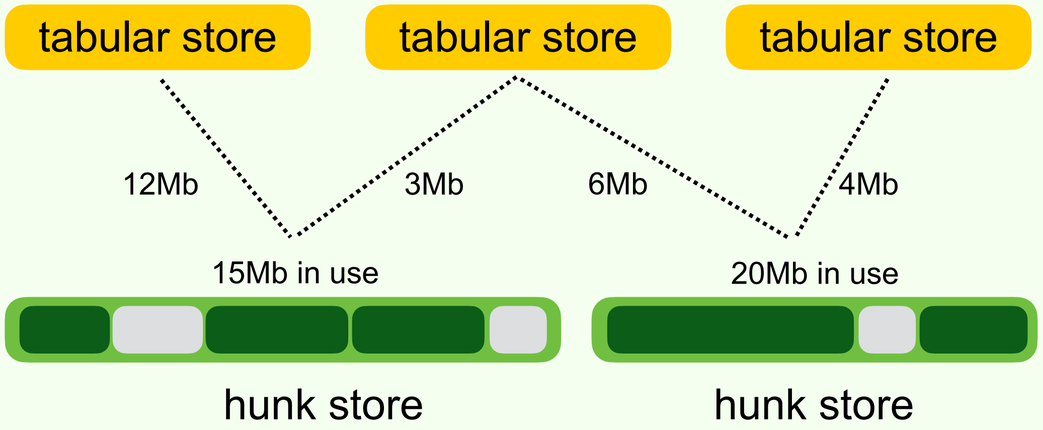
Не нужно понимать про каждый **конкретный ханк**, используется он или нет, чтобы запустить компактификацию для ханкового стора

Достаточно понимать **примерный процент** мусора в ханковом сторе

Если процент **выше порога**, то можно перезаписать ханковый стор

Взвешиваем мусор

- Двудольный граф: **верхняя** доля – табличные сторы, **нижняя** – ханковые
- **Ребро**: наличие ссылок из табличного стора в ханковый
- На ребре указан **общий объем** данных по ссылкам
- Для каждого ханкового стора можно посчитать общий объем данных, на которые **есть ссылки**



Ханковая компактификация

Доработаем табличную компактификацию

Если при табличной компактификации мы видим **ссылку на мусорный** ханковый стор, то не будем ее оставлять

Прочитаем данные из ханкового стора, **переложим** их в новый стор и поставим ссылку на него

Удаление ханковых сторов

Ханковый стор, однажды **признанный мусорным**, быстро **теряет ссылки** из табличных сторов

В конце **все ссылки пропадают**, в графе возникает изолированная вершина, удаляем ханковый стор **целиком**

Неприятности ханковой компактификации

Компактификация **перестала опираться** исключительно на **sequential IO**

Вычитывание живых ханков из мусорных сторов выглядит как **random IO**

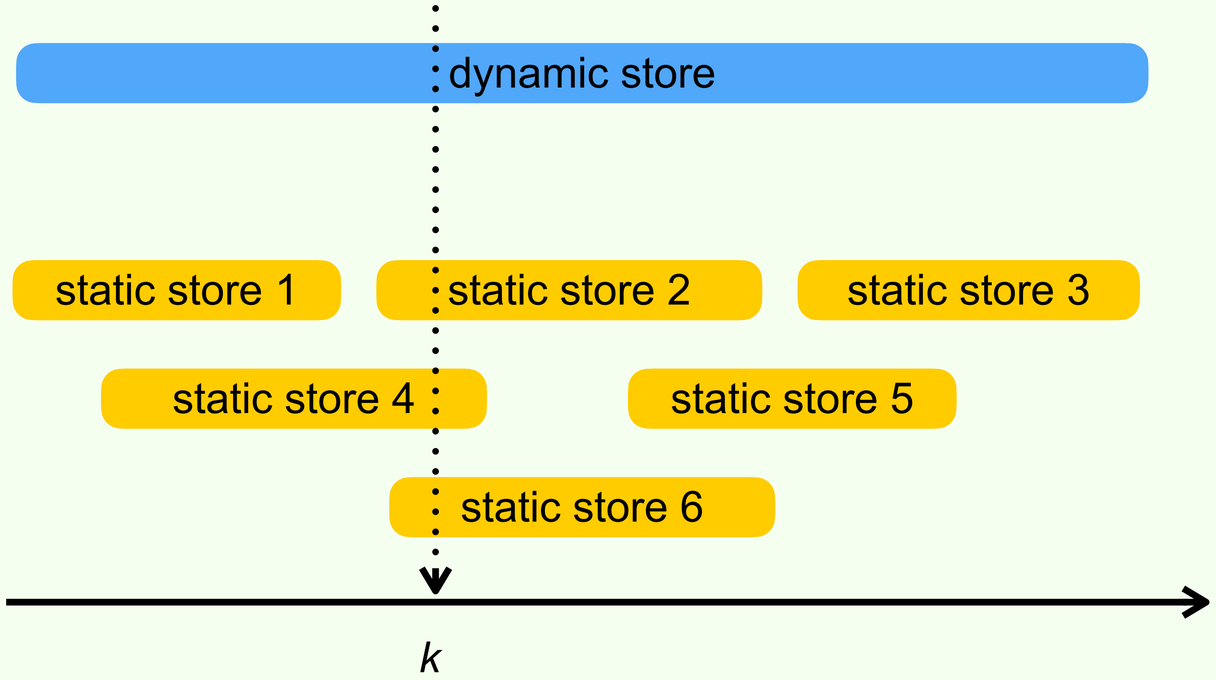
Практически **опасности нет**

При аккуратной реализации это **batch random read**

NVMe flash неплохо работает даже для весьма **мелких чтений**

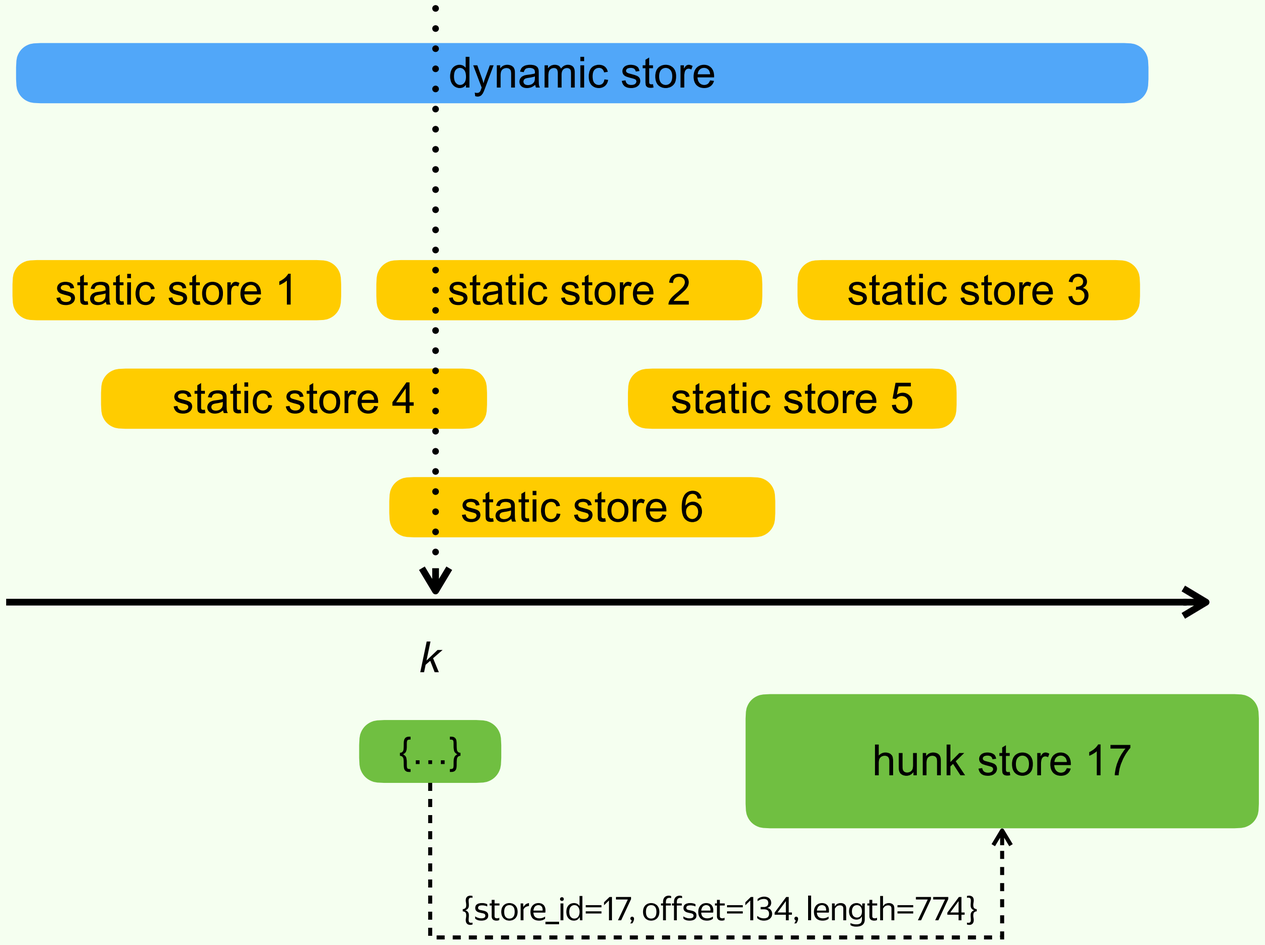
Пайплайн чтения при наличии ханков

- Читаем из табличных сторов, **разрешая конфликты** версий как и ранее
- В конце получаем либо **сами данные**, либо **ссылку** на них



Пайплайн чтения при наличии ханков

- Читаем из табличных сторов, **разрешая конфликты** версий как и ранее
- В конце получаем либо **сами данные**, либо **ссылку** на них
- Во втором случае нужен **дополнительный random read** из ханкового стора



Параметры настройки

- *M*: **минимальный размер** блоба для вынесения в ханковый стор
- *G*: пороговый **процент мусора** в ханковых сторах

Еще несколько более тонких настроек,
о которых мы не успеем поговорить :)

Ханковые трейдоффы

Минусы

Может вырасти *W:* начинаем переписывать ханковые сторы

Может вырасти *S*: откладываем освобождение места из-под мусорных ханков

Чтение требует дополнительного random read в конце

Плюсы

Падает *W:* уменьшаем число переписываний блобов при табличной компактификации

Рост *S* контролируется параметром *G*

Мелкие по размеру блобы можно не выносить в ханковые сторы, а для чтения крупных этот лишний IO не принципиален

Ищем первого заказчика

Кто использует динтаблицы в Яндексе?

Один из крупнейших клиентов динтаблиц YT – **баннерная система Яндекса** (БК)

- **Большие объемы**: сотни TB данных в каждом ДЦ
- **Высокие нагрузки**: 10M KPS случайных точечных чтений, 10M KPS записей
- Жесткие **требования по латентности**: единицы ms при p99, <10 ms в p99.9

In-memory таблицы

Исторически большинство критичных по латентности таблиц БК **поднимались в память**

В этом режиме табличные сторы пишутся на диск, но также **полностью кешируются** в RAM

Читать их с диска **не реалистично** даже при использовании NVMe flash

Чтения из памяти **малочувствительны к амплификации**, связка RAM+CPU все еще очень быстрая

Одна из самых крупных таблиц (*профили*) занимает **50 TB в памяти**, располагаясь на **сотнях хостов**

Инсталляция становится **RAM bound**

Невозможно **полностью утилизировать** CPU кластера

Пользователи **хотят расти по объему**, и без добавления новых хостов в кластер **расти им некуда**

In-memory таблицы с ханками

Включим ханки, но подгрузим в память только табличные сторы

- Уменьшим *W*, при этом *A* нам по-прежнему не важно, поскольку **мы читаем из памяти**
- В конце нужно **не более одного random read** с диска (NVMe flash)
- *S* растет, на диске становится больше мусора, но у нас **много свободного места**

Внедряем!

- **Запас места** на диске достаточный, консервативно взяли *G*=0.3
- Можно контролировать, **какая доля чтений** проливается в конце на диск, выбирая нижний размер ханка для отселения *M*
- Начали с бесконечного *M* и далее **плавно понижали** до 150
- Следили за **латентностью чтений** и снижением расходов RAM

Внедрили...?

- Сразу после включения латентность вела себя **не очень хорошо** :-(
- Возникают **дополнительные раунды** коммуникации между хостами
- Чтение с дисков намного менее предсказуемо, чем чтение из памяти

Нелегкий путь…

Проект начали в **2021**

Планировали закончить в **2022**

В реальности перевели ключевых клиентов на ханки во всех кластерах лишь в **2023**

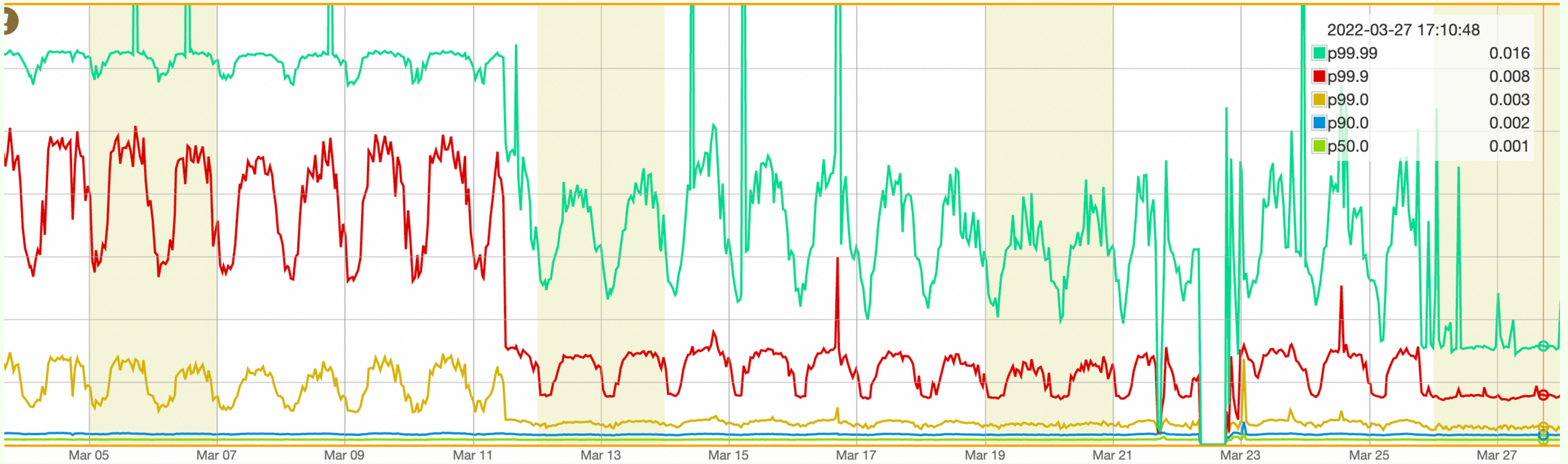
Включили **троттлеры** для фоновых активностей, чтобы не забивать сеть и диск **всплесками**

Поддержали чтение данных с NVMe flash через linux **uring API**

Реализовали **consistent hashing** для размещения реплик статических сторов, позволивший уменьшить RPC fan-out для множественных чтений

…и еще десятки мелких оптимизаций

…к успеху!

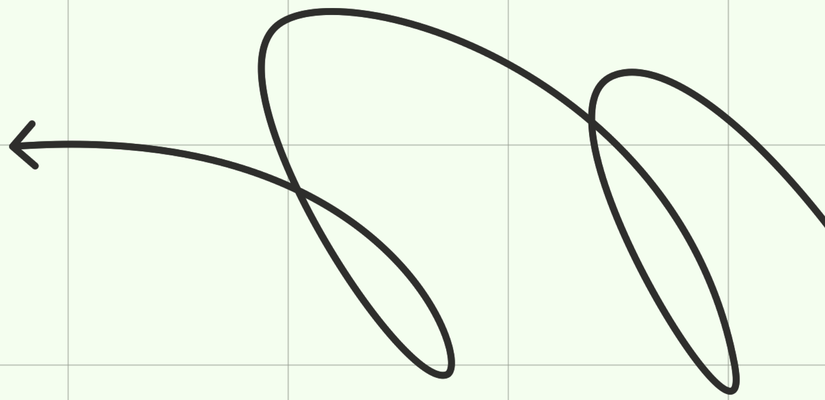

2022-03-27 17:10:48
p99.99 0.016
p99.9 0.008
p99.0 0.003
p90.0 0.002
p50.0 0.001
Mar 05
Mar 07
Mar 09
Mar 11
Mar 13
Mar 15
Mar 17
Mar 19
Mar 21
Mar 23
Mar 25
Mar 27

Итоги

- Исполнили **требования по латентности**: единицы ms при p99, <10 ms в p99.9
- **Освободили память:** вместо 50 TB таблица профилей в памяти стала весить 10 TB
- Освобожденные ресурсы **не стоят без дела**: пользователи нашли, куда употребить излишки :)

Выводы и уроки

Очевидные наблюдения



1

Собственный инфраструктурный стек – это дорого…
но одновременно и крайне полезно, когда нужно предпринять глубокие нетривиальные доработки

2

Внедрять параметрические оптимизации намного проще, чем оптимизации вида *все или ничего*

3

Нужно не забывать умножать на π все сроки :)

YTsaurus

Вопросы?

Максим Бабенко
руководитель проекта YTsaurus, Яндекс